[[harmony: block_scoped_bindings]]

ES Wiki

Show pagesource
Old revisions
Recent changes
Search

Trace: » const_functions » inherited_explicit_soft_fields » classes_as_sugar » egal » block_scoped_bindings

The new blocked scoped binding are let, const, and block functions.

Common Syntax

let, const, and block functions declarations should all be allowed in a `Program`, a `FunctionBody`, or a `Block`. However, none should be considered statements by themselves, and so cannot appear in unprotected statement context. For example:

```
   if (b) var x = 9; // legal
   if (b) let x = 9; // illegal -- must be
rejected
   if (b) { let x = 9; } // legal
```

To accommodate this, we adjust the following productions from the ES5 grammar:

```
   Block:
          { StatementList? }
   SourceElement:
          Statement
          FunctionDeclaration
```

to instead be

```
   Block:
          { SourceElements? }
   SourceElement:
          Statement
          Declaration
   Declaration:
          LetDeclaration
          ConstDeclaration
          FunctionDeclaration
```

Note that a `VariableStatement` remains a kind of statement, and so can appear in unprotected statement context, as in our first `if` example above. Since the scope of a `VariableStatement` is hoisted into the containing `Program` or `FunctionBody`, the appearance of a `VariableStatement` in statement context does not cause confusion about its semantics.

Confusingly, what the ES5 grammar refers to as a `VariableDeclaration` is neither a statement nor a declaration, but rather the name-initializer bindings that occur to the right of the `var` keyword. The grammar should use `VariableDeclarator` or something better to distinguish this non-terminal from `FunctionDeclaration`, etc.

Common Semantics

ES5/strict and ES-Harmony are lexically scoped. However, because ES5/strict does not allow a `Declaration` in a `Block`, it need only create an Environment Record on each entry to a Program, Function, or catch-clause. (Full ES5 also creates an Environment Record on each entry to a with-statement, but that need not concern us here.) For ES-Harmony, we need to create a Declarative Environment Record on each entry to a `Block` as well. The new semantics of block entry resembles the ES5 semantics for catch-clause entry (12.14).

12.1-delta Block

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *blockEnv* be the result of calling NewDeclarativeEnvironmentRecord passing *oldEnv* as the argument.
3. Set the running execution context's LexicalEnvironment to *blockEnv*.
4. Perform Declaration Binding Instantiation using the block code as described by our modified 10.5 below.
5. Let *B* be the result of evaluating *SourceElements_opt*.
6. Set the running execution context's LexicalEnvironment to *oldEnv*.
7. Return *B*

NOTE: No matter how control leaves the *Block* the LexicalEnvironment is always restored to its former state.

10.2.1-delta Environment Record

Since `ConstDeclaration`s may appear at in global code, we need to promote `CreateImmutableBinding(N)` and `InitializeImmutableBinding(N,V)` to the internal supertype, Environment Record, and provide an implementation of these methods for Object Environment Records as well. Since `let`, like `const`, has a temporal dead zone, we need the same separation between creating a binding vs. initializing for mutable bindings that we have for immutable bindings.

- We redefine the meaning of `CreateMutableBinding(N,D)` to create an uninitialized mutable binding.
- We generalize the `InitializeImmutableBinding(N,V)` method to simply `InitializeBinding(N,V)`.
- Within `SetMutableBinding(N,V,S)` we assert that the mutable binding must already have been initialized.
- We refactor all *existing* callers of `CreateMutableBinding` to ensure that the binding is initialized before it can be observed; to maintain compatibility.
- We change Object Environment Record so that uninitialized bindings are in scope without touching the bindings object, and only become backed by the bindings object upon initialization.

Table 17-delta -- Abstract Methods of Environment Records

- `CreateMutableBinding(N,D)`: Create a new but uninitialized mutable binding …
- `InitializeBinding(N,V)`: Initialize the value of an already existing but uninitialized binding …
- `SetMutableBinding(N,V,S)`: Set the value of an already initialized binding …

Table 18 becomes unnecessary

10.2.1.1.2 (Declarative) CreateMutableBinding(N,D)

The concrete Environment Record method CreateMutableBinding for declarative environment records creates a new uninitialized mutable binding for the name N. …

3. Create an uninitialized mutable binding in *envRec* for *N*.

10.2.1.1.3 (Declarative) SetMutableBinding(N,V,S)

… If the binding is an immutable binding and S is true, then a TypeError is thrown. …

4. Else this must be an attempt to change the value of an immutable binding, so if S is true, throw a TypeError exception.

10.2.1.1.7 (Declarative) CreateImmutableBinding(N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new uninitialized immutable binding for the name *N*. A binding must not already exist in this environment record for N.

10.2.1.1.8 (Declarative) InitializeBinding(N,V)

… An uninitialized binding for *N* must already exist. …

2. Assert: *envRec* must have an uninitialized binding for *N*.

…

4. Record that the binding for *N* in *envRec* has been initialized.

10.2.1.2.2 (Object) CreateMutableBinding(N,D)

The concrete Environment Record method CreateMutableBinding for object environment records creates a new uninitialized mutable binding for the name *N*. If Boolean argument *D* is provided and has the value true the new binding is marked as being subject to configuration.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create an uninitialized mutable binding in *envRec* for *N*.
4. Record that the newly created binding is to be writable.
5. If *D* is true record that the newly created binding is to be configurable; else non-configurable.

10.2.1.2.3 (Object) SetMutableBinding(N,V,S)

Between 1 and 2. If *envRec* has an uninitialized binding for *N* and *S* is true, throw a ReferenceError exception.

10.2.1.2.4 (Object) GetBindingValue(N,S)

Between 1 and 2. If *envRec* has an uninitialized binding for *N* and *S* is true, throw a ReferenceError exception.

10.2.1.2.5 (Object) DeleteBinding(N)

Between 1 and 2. If *envRec* has an uninitialized binding for *N* and *S* is true, throw a ReferenceError exception.

New 10.2.1.2.7 (Object) CreateImmutableBinding(N)

The concrete Environment Record method CreateImmutableBinding for object environment records creates a new uninitialized immutable binding for the name *N*.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create an uninitialized immutable binding in *envRec* for *N*.
4. Record that the newly created binding is to be non-writable and non-configurable.

New 10.2.1.2.8 (Object) InitializeBinding(N,V)

The concrete Environment Record method InitializeBinding for object environment records creates in an environment record′s associated binding object a property whose name is *N* and initializes it to *V*. A property named *N* must not already exist in the binding object. On success, it drops its own separate record that *N* is uninitialized.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Assert: *envRec* currently records *N* as uninitialized.
3. Let *bindings* be the binding object for *envRec*.
4. Assert: The result of calling the [[HasProperty]] internal method of bindings, passing *N* as the property name, is false.
5. Call the [[DefineOwnProperty]] internal method of *bindings*, passing
    - *N*,
    - Property Descriptor { [[Value]]: *V*, [[Writable]]: the recorded writability of *N*, [[Enumerable]]: true, [[Configurable]]: the recorded configurability of *N* },
    - and true

as arguments.

Open question: Parity error applies to the last `true` argument above. If we adopt my proposed #2, then it should instead be S.

More questions:

- The global object may have arbitrary properties, so how can step 4 above assert that [[HasProperty]] returns false?
- How do `let` not in a block statement and `var` interact in global code? Are both allowed to declare the same name? If so, does the `let` binding shadow the `var` one?
- How do `let` not in a block statement and `var` interact in function code? Currently (ES5 and older), `var a;` in `function f(a) {...}` restates the argument binding.

We should forbid `let` and `var` binding the same name in the same scope. Sorry if I missed that in this strawman.

— Brendan Eich *2010/06/09 14:54*

We should also forbid `var`-declarations that have already been “shadowed” by `let`-declarations, such as:

```
var x = "outer";
function f() {
    {
        let x = "inner";
        {
            // after hoisting, the initializer mutates the let-binding!
            var x = "sneaky"; // this should be illegal
        }
        print(x); // prints sneaky!
    }
    return x; // prints undefined!
}
```

This might be implied by Brendan’s last statement, but I’m just clarifying that it’s more than just forbidding `let x; var x;`.

— Dave Herman *2010/08/20 16:33*

10.5-delta Declaration Binding Instantiation

After step 1. Let *lexEnv* be the environment record component of the running execution context’s LexicalEnvironment.

Step 5-delta. Change all uses of *env* to *lexEnv*. Generalize this to apply to each *`Declaration`* in code, rather than just each *`FunctionDeclaration`*. Rephrase to avoid `Declaration`s in nested blocks and catch-clauses. Refactor so that the particulars for each kind of `Declaration` are defined by that `Declaration`.

Step 8-delta. Rephrase to make clear that this step is skipped on entry to blocks and catch-clauses, and that the enumeration of `VariableDeclaration`s in the remaining cases must traverse into nested blocks and catch-clauses.

harmony/block_scoped_bindings.txt · Last modified: 2010/08/20 16:52 by dherman

Show pagesource
Old revisions
Login
Index
Back to top